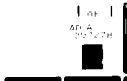
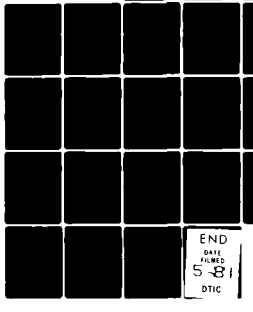
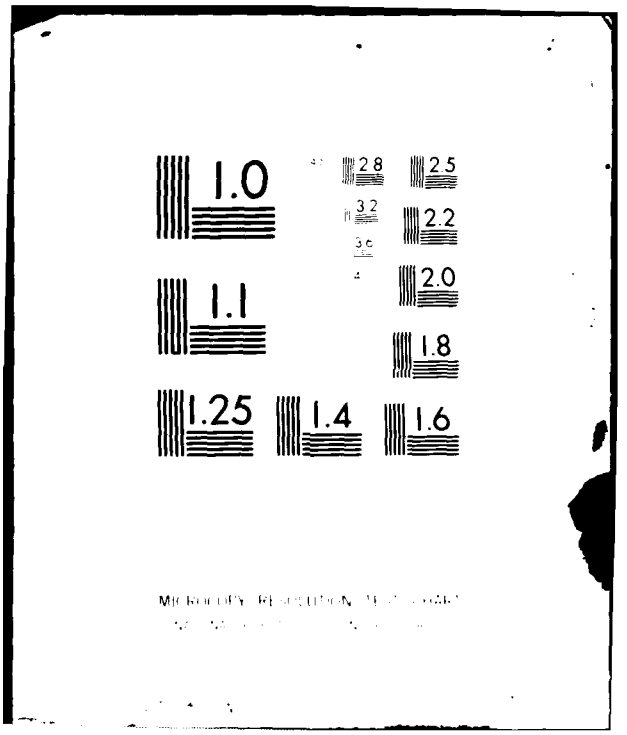
AD-A097 278 MARYLAND UNIV COLLEGE PARK DEPT OF COMPUTER SCIENCE F/G 9/2
SOFTWARE REQUIREMENTS: A REPORT ON THE STATE OF THE ART.(U)
OCT 80 R T YEH, P ZAVE, A P CONN, G E COLE AFOSR-77-3181
AFOSR-TR-81-0320 NL

UNCLASSIFIED





END
DATE FILMED
5-81
DTIC


1.0
2.8
2.5
3.2
2.2
3.6
2.0
1.1
1.8
1.25
1.4
1.6

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

LEVEL II

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: AFOSR-TR-81-0320	2. GOVT ACCESSION NO.: AD-A097278
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): SOFTWARE REQUIREMENTS: A REPORT ON THE STATE OF THE ART	5. TYPE OF REPORT & PERIOD COVERED: INTERIM
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Raymond T. Yeh, Pamela Zave, Alex Paul Conn & George E. Cole, Jr	8. CONTRACT OR GRANT NUMBER(s): AFOSR-77-3181
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Department of Computer Sciences, University of Maryland, College Park MD 20742	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 2304/A2, 61102F
11. CONTROLLING OFFICE NAME AND ADDRESS: Air Force Office of Scientific Research/NM, Bolling AFB DC 20332	12. REPORT DATE: OCT 80
	13. NUMBER OF PAGES: 54
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Interim technical rept.	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

DTIC ELECTE APR 2 1981 S D B

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Many problems arising during the development and evolution of large computer systems can be alleviated by thorough and timely attention to requirements. This paper surveys recent research on requirements technology which the authors consider to promise substantial improvements over the analysis and specification techniques that are now commonly known and used. All aspects of requirements are considered, and points are illustrated using the requirements document of an early, but ambitious, real-time system.

DD FORM 1 JAN 73 1473

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

409022

AD A097278

COPY

TR-949 October 1980

SOFTWARE REQUIREMENTS: A REPORT ON THE STATE OF THE ART

Raymond T. Yeh, Pamela Zave,
Alex Paul Conn, and George E. Cole, Jr.

AFOSR-77-3181

Abstract

Many problems arising during the development and evolution of large computer systems can be alleviated by thorough and timely attention to requirements. This paper surveys recent research on requirements technology which the authors consider to promise substantial improvements over the analysis and specification techniques that are now commonly known and used. All aspects of requirements are considered, and points are illustrated using the requirements document of an early, but ambitious, real-time system.

AIR FORCE OFFICE OF SCIENTIFIC RESEARCH (AFSC)
NOTICE OF TRANSMITTAL TO DDC
This technical report has been reviewed and is
approved for public release IAW AFR 190-12 (7b).
Distribution is unlimited.
A. D. BLOSE
Technical Information Officer

1. Introduction

Statistics gathered during the past few years have produced an alarming awareness of the enormous cost of maintaining large software systems. If the trend continues, the data-processing industry not only will become the most labor-intensive industry, but also will devote most of its productivity to maintaining old, ill-structured, and difficult-to-modify software.

Furthermore, it has been shown ([Belady & Lehman 79]) that as the complexity (and entropy) of a system grows, the probability increases that a change will introduce additional errors. The result is an increasingly unreliable system. Real danger is involved in the dependence of our society on such systems, as illustrated recently by false alarms triggered by software errors at the Strategic Command Center (reported in the Washington Post).

Although there are many reasons for the difficulty of maintaining software, lack of thorough attention to requirements analysis and specification, the earliest phase of software development, is a major one. For example, in two large command/control systems, 67% and 95% respectively of the software had to be rewritten after delivery because of mismatches with user requirements ([Boehm 73]). There are also many examples of total cancellation of projects due to lack of appropriate requirements and feasibility analyses. Some of the more expensive cases are the $56 million Univac-United Airlines reservation system and the $217 million Advanced Logistic System ([Boehm 80]). In general, it has been found that "design errors" (all errors made before implementation) range from 36% to 74% of the total error count ([Thayer 75]). These

numbers are not the whole story, however--a design error takes 1.5 to 3 times the effort of an implementation error to correct.

We have illustrated the importance of developing a good requirements methodology to control maintenance costs, but there are other equally pressing reasons. The requirements document has a unique role in the development of a software system: it is the basis for communication among customers, users, designers, and implementers of the system, and it is unlikely that the project will be a success unless it represents an informed consensus of these groups. It must also carry the weight of contractual relationships between parties that sometimes become adversaries; in particular, the design and implementation must be validated against it.

The costs of neglecting these functions include lack of management control, inability to use top-down design or other software engineering techniques, user hostility, and lawsuits. In short, because the requirements phase comes so early in development, it has tremendous leverage in the quality (or lack thereof) of the development effort and the final product.

Current approaches to requirements engineering, unfortunately, are quite inadequate. Most of the available techniques concentrate on functional requirements, and provide relatively weak structures for expressing them. They offer basically tools (primarily languages), rather than guidelines for analysis or specification.

In this paper, we suggest a systematic approach to obtaining software requirements, and point out the existence of available results from other fields such as database management, artificial intelligence, and psychology that are of great relevance to the development of a good

requirements document. We deal with all aspects of requirements documents, and illustrate them with examples taken from an existing requirements document (see below). Due to space limitations, our discussion will be largely informal, but will guide the interested reader to more thorough presentations elsewhere.

The AFWET system (Air Force Weapons Effectiveness Testing, ultimately realized under the name "WESTE") was an early real-time system which supported quantitative testing of U.S. military (conventional warfare) capability (see Figure 1). We describe it briefly here because its requirements document ([Air Force 65]) is a fruitful source of bad examples and unsolved problems.

Tests were military exercises involving "test elements" such as airplanes, ships, tanks, and ground defense positions (some playing the role of enemy forces), confined to a circle centered on Eglin Air Force Base in Florida. Test elements communicated with a central site through military standard radio equipment, plus a contractor-supplied communications network.

During the test, moving elements would send periodic notifications of their positions to the central site. Mock firings of weapons would also cause messages to be sent, supplying all relevant parameters such as the direction of aim. The central system would simulate the battle in real time, determining which of the mock firings would have resulted in "kills". The results of the simulation were (a) used to display the course of the battle on graphics scopes for the benefit of officers in a control room, (b) dumped onto archival storage for later analysis, and (c) used to send "kill" notifications to "killed" test elements in the field. They would then react with a flashing light or loud noise, and cease to participate in the battle.

2. Conceptual Modeling

In the early days of software development, machines were relatively small and so were systems. A program served in a well-understood, well-specified scientific domain, and thus could be written directly from a statement of need.

As we have moved to much larger systems and a variety of application domains, the need for precise specification of a system before implementation has increased. But the complexity of these systems demands an additional layer of understanding, a "buffer", between the real world and the requirements specification. This "buffer" allows an analyst to understand the problem before he proposes a system to solve it--an understanding that can be achieved with an unassisted mental model, if the problem is simple enough. For complex problems, a model must be constructed which is explicit and formal enough to be shared by a group of people. We call such problem models conceptual models because they are constructed at the level of human concept formulation.

One possible consequence of the lack of a conceptual model appears in the AFWET requirements:

> Choice of major subsystems shall be the responsibility of the contractor; however, a typical range configuration may consist of the following subsystems....Space Position Subsystem...Data Subsystem...Timing Subsystem...Communication Subsystem...Processor Subsystem...Kill and Display Subsystem... .

For lack of an approach to providing an introduction to, or overview of, the AFWET problem, the requirements writers had to present part of a design for the system!

If we accept the assumption that constructing a conceptual model is a necessary step in gaining understanding of large, complex problems,

what do we model and how do we construct it?

We believe that conceptual modeling should be done "outside-in", beginning with the proposed system's environment and working inward toward the system. In many cases this will lead the analyst directly to the requirements, since the purpose of the system is to support a desirable mode of operation in the environment. This is particularly true when the project is to automate existing manual procedures, because then the computer system is a direct reflection of the current operations.

Other reasons for stressing understanding of the environment are that it will improve communication with customers and users (who are much more interested in their environment than your system), and because large application programs are parts of their own environments ([Lehman 80]). But possibly the most important of all, given the intrinsically evolutionary nature of large software systems ([Belady & Lehman 79]), is that change in a system originates with change in the environment. By modeling the environment, the analyst can study potential changes, and possibly even provide a designer information that can be used during design so as to acheive modularity--the property that small changes in the environment cause correspondingly small changes in the system.

The overall structure of a conceptual model is shown in Figure 2. The environment consists of identifiable objects such as people, airplanes, terminals, forms and other types of data, etc. The states of these environment entities must be represented, as must the events (an agent makes a flight reservation, a machine overheats) which cause state changes. The target system can be similarly divided into a state, and activities which interact with environment events so as (ultimately) to influence them.

The model is structured by relationships and constraints on all these objects. "A is a subnet of B", "Helen and Bob are married", "faucet must be opened before water can flow", etc., are simple examples of constraints and relationships, but new government regulations, hardware configuration changes, and a wide variety of other facts can be relevant.

To collect information on the environment, personal interviews and questionnaires are most often used. The actual modeling process can begin with either entities or events. When starting from an event, the information change due to occurrence of that event must be reflected into the structure of the model. For example, a transaction "reservation request" from a remote terminal linked to an airline reservation system will change the available seats on a particular flight. Assuming such information is stored in a target system database, this event will also trigger a system activity to change the database. The state information is thus transmitted through the interaction between the system and its environment. Similarly, an analyst may start with entities from the environment to build up state information, and then consider changes to these entities so as to develop the structure of processes within the system. Interested readers are referred to [Yeh et al. 79] for more details about information collection, and to [Yeh & Zave 80] for more examples of conceptual modeling.

Note that the "outside-in" approach is neither "top-down" nor "bottom-up"--the model structure must be evolved in both directions. Top-down analysis is employed when an analyst asks an interviewee to elaborate on some previously identified feature of the environment, but a bottom-up approach is required to collect and integrate views of

users in different parts of an organization.

The conceptual model is an important tool for understanding the requirements of a system, but it is not a requirements document. The latter must be derived from the model, but has somewhat different properties. This will be the subject of the next section.

3. Requirements Derivation

The conceptual model should be a rich, complex information structure--probably too much so for the purposes of the software requirements document (SRD). For instance, a conceptual model of AFWET might include views of the system as seen by soldiers, officers, computer operators, and hardware maintenance personnel, and thus be highly redundant. It might also model more of the environment than is needed just to define the proposed system. This would be the case if the AFWET analysts decided (quite correctly!) that they needed to understand the military background and purpose of the tests they would be instrumenting, to assure themselves that the data they gathered and displayed would be useful.

Thus the SRD is derived from the conceptual model by filtering and organizing, constantly aiming towards the "best-engineered" specification. Explicit goals for the SRD can be found by considering the thing it will be used for:

Many groups of people must communicate with each other through and about the SRD. Therefore it must be understandable.

In order to accomodate the evolutionary nature of large systems, the organization of SRD must be structured so that changes can be made with minimum effort. In a word, it must be modifiable.

Last but not least, the SRD is used to define the target system. To do this properly it should be precise (preferably formal), unambiguous, complete (a particularly important aspect of completeness being the role of the SRD in contractual obligations), internally consistent, and minimal. A minimal specification does not over-constrain the design

of the system, which might exclude the best solutions to design problems.

As in most engineering situations involving multiple goals, the above list of properties cannot all be achieved in most situations. However, there does exist a set of mental tools and principles which can help the analyst to meet many of them.

The crucial issue here is decomposition of complexity, also referred to as "separation of concerns", "divide and conquer" strategy, etc. For software development we can describe the goal of this principle via two subgoals, namely: the "process goal" and the "product goal". The process goal is to keep the process under our intellectual control at all times. The product goal is to organize the product in a fashion that allows others to comprehend the product by an amount of effort which is proportional to the size of the product. There are three powerful tools for decomposing complexity so as to achieve these goals (see Figure 3).

The first tool is the notion of abstraction. The use of abstraction allows us to suppress details and concentrate on essential properties. Thus, we refer to something as an abstraction if it represents several actual objects but is disassociated from any specific object. The use of abstraction forms natural hierarchies, allowing elaboration of more and more detail and hence providing intellectual control of the process.

The second tool is partition, i.e., representing the whole as the sum of its parts. This tool allows us to concentrate on components or subsystems of the system one at a time. Partitioning makes systems modular. Note that if each partitioned component also has an abstraction hierarchy, then we have both a horizontal and a vertical decomposition of the system.

The third tool is projection, and enables us to understand a system from different viewpoints. A projection of a system represents the entire system, but with respect to only a subset of its properties; the perfect physical analogy is an architectural drawing, which is a two-dimensional view of a three-dimensional building. The notion of a "view" of database [Astrahan et al. 76] is another such example. Again, this tool allows us to separate particular facets of a system from the rest and therefore retain intellectual control.

Of course, any tool can be abused. In using the three structuring principles above, one must be guided by the principal of "information hiding" ([Parnas 72b]) and other observations about how specifications can be made coherent and flexible ([Parnas 79]). Arbitrary decompositions forced on system aspects which are too interdependent will cause more problems than they solve.

4. Functional Requirements

Functional requirements describe what the target system does, and are clearly the heart of the SRD. Section 2 introduced the notion of an explicit model of the proposed system's environment, as an important tool in requirements analysis (leading to the global model shown in Figure 2). From the viewpoint of functional requirements specification, there is an important additional advantage to having an explicit model of the environment: since interactions between the environment and target system can then also be made explicit, it is much easier to specify the all-important environment/target system interface in an accurate, precise, understandable, and yet modifiable manner.

Thus the task of functional requirements specification is to find a formal representation for the detailed information needed to fill in Figure 2. The major challenge is complexity, and we will classify approaches to specification according to the primary dimension along which they decompose structure.

4.1. Data Models

Data-oriented models concentrate on specifying the states of Figure 2--the state of the target system will always be represented as a data structure, and the state of the environment can be modeled as such, even though the resulting data structure need never be implemented.

Research on database problems has led to the recognition of abstract concepts which can be fruitfully used in data-oriented specifications of software systems of all kinds. [Smith & Smith 79] is a survey aimed at non-database-specialists, and we shall use the notation presented there for explaining the most prominent data-structuring concepts.

At any moment, a database or "data space" consists of a population of "individuals" or items. A database modeling the environment of the AFWET system ("STATE of entities in environment" in Figure 2) could contain individuals representing test elements.

Individuals belong to (are "instances of") types, and types can be subtypes of other types (obviously, an instance of a type is also an instance of all its supertypes). The type hierarchy for a database is specified as part of the database's "type definition". Table 1, for instance, is a type definition for the real-time simulation portion of the AFWET system. It defines types using the syntax:

```
def   TYPE-NAME:
      :
      :
      end,
```

and the types listed after the keyword "sub" are its subtypes. Thus an individual plane may be a member of types "B-52G", "PLANE", and "TEST-ELEMENT". The use of types to structure data is referred to as generalization.

Individuals also have components which are needed to describe them fully. Each component is identified with a type to which it must belong, and the proper components of individuals of certain type are listed after "com" in that type's definition. Thus full description of an "air-position" (instance of type "AIR-POSITION") requires a "surface-position" and an "altitude". Full description of a "plane" requires its "air-position" and the "weapons" it is carrying. It also requires a "role" (friend or foe), a component that the "PLANE" type inherits from its supertype "TEST-ELEMENT". It is clear that the component relation is also hierarchical, and its use to structure data is referred to as aggregation.

The state of the AFWET system (real-time simulation portion only)

and its environment, at any given time, is a database consistent with this type definition (plus a great deal of read-only information, such as models of test-element motion and weapons threats, needed for simulation). There should be only one instance of type "BATTLEFIELD", and its components (* means that the individual can have multiple components of the designated type) represent the currently active "test-element"'s. Within the system, the state of the battlefield at a given time is represented by an instance of type "FRAME", having as components a "time" and the "test-element"'s that were active at that time. A full "test" is recorded in multiple "frame"'s. This is a particularly good example of what it means to reflect the structure of the environment within the system!

A data model must be interfaced with processing aspects of the system. At the very least, a set of primitive data manipulation operations should be enumerated, and these operations should be defined in terms of a first-order predicate language and the operations create, destroy, and modify applied to individuals in the database. For instance, in the AFWET database a simulated kill should destroy a "test-element" that is a component of the current "battlefield". Once defined, these operations can be used as the interface between the data model and whatever higher-level processing model is preferred (see 4.2, 4.3).

Data-oriented models, as the heart of requirements analysis and specification, have been very successful--especially in the domain of data processing and business information systems. Some good examples can be found in [Yeh et al. 79], [Roussopoulos 79], and [Mittermeir 80]. The primary notion used there is the semantic net, a graphical formalism which was originally developed by artificial intelligence researchers for representing knowledge structures.

4.2. Dataflow Processing Models

The most common model of processing used in conjunction with data-oriented models is the dataflow diagram, which simply names the major processing activities of the system, and indicates which parts of the data model are inputs and outputs to each activity. If iterative refinement of the dataflow diagram is supported, and activities are defined (usually informally) in terms of data manipulation operations, a level of expressive power sufficient for many data-processing systems may be achieved.

The dataflow approach is central to SADT ([Ross & Schoman 77], [Ross 77]), although there is additional emphasis on a methodology for team cooperation. The dataflow approach is supported with automated tools in PSL/PSA ([Teichroew & Hershey 77]). It can be extended with control information via Petri nets ([Peterson 77]), or with resource synchronization ([Conner 79]).

The deficiencies of the dataflow model for specifying embedded (real-time) systems are apparent in the dataflow diagram for the AFWET system shown in Figure 4 (the "fight" function has been added to provide the "processing" that modifies the state of the environment). The global events and activities in this system are continuous, and are not activated by the appearance of a single input, or any simple combination of inputs. At a lower level, they consist of complex combinations of pieces of computation which must occur asynchronously and in parallel. Dataflow as a concept is simply not powerful enough to permit precise specification or effective decomposition of systems, such as embedded ones, in which concurrent and asynchronous operations occur at the requirements level.

4.3. The Process Model

An approach which is better suited to specification of embedded systems emphasizes the "events" and "activities" portions of Figure 2. The central concept is the process, an autonomous computational unit which is understood to operate in parallel with, and interact asynchronously with, other processes. Processes have long been used as abstractions of concurrent activity within multiprogramming systems ([Horning and Randell 73]), and many recent articles have shown that they can be used to model databases, monitors, functional modules, I/O devices, and presumably any other identifiable structure within a computing system (e.g. [Hoare 78], [Brinch Hansen 78]).

Formally, a process is just a "state space" (set of possible states) and a "successor relation" which maps predecessor states onto their possible successors. This simple concept is easily adapted to being a digital simulation of an object (person, machine, sensor, etc.) in the environment of a computer system.

The result of the generality of processes is that the requirements for a system can be specified by a set of asynchronously interacting processes, some of which represent objects in the environment, and some of which represent objects in the target system. The environment of AFWET, for instance, becomes a set of processes, each one simulating a test element. All processes respond to kill messages by becoming inactive. All processes representing test elements with weapons send firing messages whenever their internal, cyclic, simulation algorithms decree that they have fired a weapon. All processes representing moving test elements periodically update their positions and send position messages to the central system. The result is an easily constructed, easily understood

model with highly complex overall behavior; it is understandable simply because it is naturalistic, being made up of semi-autonomous objects acting in parallel, just as the real world is.

Figure 5 shows an overall process-and-communication structure for the simulation portion of the AFWET system. The test-element processes are as described above. Processes representing radio towers put timestamps on the input messages and relay them to an input-buffer process. The input-buffer process collects into a batch all the messages relating to the period covered by a particular simulation step, waits until all messages from that period can reasonably be expected to have arrived, and then passes the batch to the simulation process. This simulator computes a new frame (and kill messages) from the old frame, the batch of messages, and its various mathematical models. Frames are passed on one by one to the rest of the system, where they are stored and otherwise used. Kill messages are relayed to the test-element processes via an output-buffer process and the radio-tower processes.

This has been an informal description of a formal requirements specification in PAISLey (Process-oriented, Applicative, Interpretable Specification Language, see [Zave 79a], [Zave 79b], [Zave & Yeh 80], [Zave 80]). PAISLey is a process-oriented language aimed at embedded systems. As the acronym indicates, its other major characteristics are that it is applicative and interpretable. The advantages of applicative languages are currently receiving well-deserved attention, and recent results and trends are surveyed in [Smoliar]. The most important properties of applicative languages (for our purposes) are that they are precise and convenient vehicles for abstraction, and that they are interpretable.

Interpretability carries with it many advantages. It means that the

system-plus-environment model is executable, and that it can be validated by testing--including demonstrations of behavior for customers, and performance simulations (if necessary). The advantages also continue throughout development, since the environment part of the model can be used as a "test bed" or "driver" during development, and the model of the proposed system can be used as the standard for acceptance testing.

The use of an excutable model which emphasizes the active parts of the required system can be termed the "operational" approach. The operational approach was first taken by the SREM system and its requirements language RSL ([Bell et al. 77], [Alford 77], [Davis & Vick 77]). In RSL processing paths from input stimulus to output response are specified directly, and can be simulated for performance purposes. Stimulus-response paths are an important aspect of operational requirements, of course, but are incomplete in not including explicit representations of system states, internal synchronization, or potentially distributed environments.

PAISLey is more complete than RSL in including states, synchronization, and the system's environment, and it shows that the operational approach has some striking advantages for embedded system requirements.

One advantage is that the rigor of having to make a model that "runs" always proves to be a powerful influence against ambiguity and vagueness in requirements. In the AFWET example, for instance, the relationship among frames, input messages, and real time was arrived at after a great deal of confusion. It finally became clear that: (a) simulation had to be oriented toward increments of time rather than toward events, because the effect of a "firing" event may occur at any point during the entire interval that the bullet is still in the air; (b) the cost of backtracking would be prohibitive--once a computation was done it could not be undone

by a late-arriving message; and (c) this meant that simulation time had to be enough behind real time so that all messages relating to a new frame could be assumed to have arrived when computation of the new frame began. Timestamps are put on messages at the radio towers because there is variable delay in the rest of the communication network, and the simulator must know accurately to which time a message refers (delay in the radio communications, being nearly constant, is not a problem).

Another advantage of operational specifications is that they provide natural structures to which performance requirements can be attached (this is especially important for embedded systems, given the prominence of performance in that domain). [Zave 79b] and [Zave & Yeh 80] show how response time and feedback loop requirements can be specified formally, simply by attaching timing attributes to functions in the specification. In Figure 5, the timing requirements are more complex, but still representable in the same formalism. If g is the "granularity" of the simulation, i.e. the increment of time between frames, then the successor function of the simulation process (which computes the next frame) must never take longer than g to evaluate. If the delay in radio communications is r, and the simulation time is to be no more than $r + d + g$ behind real time, then d must be the upper bound on delay in getting messages from the radio towers to the input buffer. Finally, if the timestamps are to be useful, the upper bound on the time it takes for any process to read the real-time clock must be very small.

A third advantage is that operational specifications make it possible to include resource requirements when necessary. Resource requirements are requirements that a particular resource or quantity of a resource be used. In AFWET the use of a time-multiplexed, fixed-delay radio communi-

cation link was a resource requirement*, imposed because that equipment was already owned and installed. PAISLey offers the generality of a complete model of computation, including asynchronous distributed computation. This means that no new system problem will surprise us with concepts unexpressible in the language.

Observations about performance and resources bring us to everyone's major reservation about operational requirements: Aren't these actually design specifications? Don't they say much more than should be said in the requirements? We believe that the answer is no, for the following reasons.

Extensive experience with requirements examples shows clearly that the essence of true design is managing scarce resources to meet performance goals. As long as a formalism does not force the specifier to make unnecessary decisions about performance and resources, then it is not forcing him to specify design rather than requirements. Fortunately, both applicative languages and the process model are excellent in this respect. For the "design-independence" of applicative languages, see [Backus 78] and [Smoliar 79]; for the "design-independence" of processes, consider this very basic example: In Figure 5, we used as many processes to describe the central site as were logical and convenient. The design for this system will look quite different, since it will probably have to deal with a scarce resource problem--only one processor. The designer will have to determine how a single processor can be multiplexed so as to implement, and meet the performance requirements of, all processes at the central site.

Another example of how the specification described by Figure 5 differs

---

* Actually this is an inference from the requirements document, which is by no means clear on this point.

from a design concerns the real-time clock process, which "ticks" at regular intervals and can be read by other processes. It is not technically feasible to build a single global clock which can be read fast enough by a collection of remote sites. The design to meet this performance requirement will probably entail local clocks (which can be read fast enough) and a global synchronization protocol executed before each test begins.

The existence of resource requirements forces us to recognize two distinct meanings of "design": from the "technical" viewpoint, it is managing resources to meet performance goals, but from the "administrative" (economic, political) viewpoint it is any property of the system that is not required to satisfy whoever is paying for it. From the technical viewpoint resource requirements are premature design decisions, but from the administrative viewpoint they are common and entirely legitimate requirements. They should be minimized, but can never be eliminated, and any requirements language that cannot handle them will be inadequate in many situations.

To summarize, operational structures do not over-constrain design unless they select a particular solution to a problem which has other feasible solutions. The AFWET example is rather extreme in that it involves quite a bit of what is technically design detail, but all the "design" decisions were forced into requirements analysis by the customer's wishes or the technical infeasibility of other approaches.

A final, but important, property of any activity-oriented requirements language is its interface with data-oriented specifications. Different as Figure 5 may seem from the data model of 4.1. the two are actually quite compatible. The components of a "test-element" in the

data model are exactly the same as the components of the state of a test-element process in the process model (see Figure 6)! This means that the process model and the data model can be viewed as projections of the same underlying model, which contains both, and the proposed system model supports a similar decomposition. Given the data and process complexity found in most large systems, compatibility between appropriate models for them cannot receive too much attention.

5. Nonfunctional Requirements

Requirements other than functional ones have received very little attention, but may be an equally important part of the SRD. Since the state of the art is very far from having a comprehensive theory or methodology for these requirements, we present an annotated outline, intended to be used as a checklist of the various topics that should be at least considered, even if not included in the SRD. (Human factors have been omitted, because they are discussed at length in the next section.)

I. Target system constraints

IA. Performance

Performance is defined here to include all factors which describe both the subjective and objective qualities of the target system. It is thus a measurement of the "success" of the target system, a constraint below which the system must not be allowed to fall.

IA1. Real time

In many systems, especially those which are embedded in or connected to specialized equipment, the real-time performance is essentially a measurement of the success of the system. For example, in AFWET, if the system is not designed in some way to accept weapons firings in real time, then it is likely that some of the firings will be entirely missed and the validity of the mission will be compromised.

IA2. Other time constraints

"Other time constraints" refers to important relative timing considerations within the target system, which relate events to each other rather than to real time. These relationships would normally involve precedence but might also include information for choosing between competing activities based on some kind of priority system. In AFWET, for example, the computation of the lethality of a missile might not be carried out until the trajectory has been determined, and both the computations may be considered more important (of higher priority) than the movement of an unrelated tank, given a scarcity of computational resources. While detailed decisions on precedence or priority throughout the target system may be left to the designer, there should be a means for including critical constraints in this area in the requirements.

IA3. Resource utilization

Closely related to the rationale for assigning precedence constraints are the constraints on resource utilization. A system will often be built in which the computer resources are attached to expensive specialized equipment. Decisions on which equipment to service in which order may very well be directly related to the cost or importance of each item of specialized equipment, and the performance of the system may very well be assessed in terms of the response to the needs of this equipment. The requirements thus should provide a means for specifying the handling of critical equipment and also for placing a constraint on the balanced or optimal use of the remaining resources in the system.

IA4. Accuracy, quality, comprehensiveness

While timing and resource utilization are fundamental aspects of performance, other factors also characterize the target system performance. The accuracy of the detection and computation of data can be critical. In AFWET, two elements in combat must be accurately tracked for position, both by information that might be transmitted by the elements themselves or by appropriate external equipment, such as radar. This position information must be maintained at the same degree of precision during computations or the determination of a "kill" condition might be erroneous.

The quality of the target system can be an important requirement. For example, if a CRT display is fuzzy or distorted, then the relationships between elements might be incorrectly interpreted by viewing personnel. Akin to quality is the idea of comprehensiveness. If important data that could be displayed is never made available, or not presented when it could be a determining factor in a mission, the target system is performing at a less-than-optimal level. The requirements constrain the eventual design by identifying, at least in general terms, the degree of comprehensiveness desired. Designers can later figure out how to manipulate the data within the quality and human factors constraints.

IB. Reliability

The reliability portion of the requirements outline has been adapted from the presentation in [Jones & Schwarz 80], a study of multiprocessor systems. Reliability can be divided into two basic categories: availability of the physical equipment and integrity of the information. For requirements, the concern is about "failures" (noticeable events where the system violates its specifications) rather than "faults" (mechanical or algorithmic defects which will generate an error) ([Zelkowitz et al. 79]), since the means by which the system maintains its specified level of reliability is a concern of the system developers, not those who write requirements. The purpose of defining and classifying the failures is so that constraints can be placed on the likelihood of such failures.

IB1. Availability

Failures that affect the availability of the system are, in general, ones that cause one or more devices to cease to function. Occasionally, a device will continue to operate, but at a reduced speed. If equipment ceases to function, we are not only interested in the duration of such failures, but also, the actual impact on the system. Today, it is much more common for critical components to be replicated and interconnected in such a manner that the target system continues to operate under a wide variety of failures. The requirements will have to be able to address the extent of degradation permissible under specified failures and the means with which the system copes with these problems (e.g. manual versus automatic reconfiguration). In a system such as AFWET, the ability to transmit critical information over more than one communication channel might be a requirement for dealing with a failure of one of those channels.

IB1a. Definition and classification of failures that cause degraded functioning

IB1b. Probability of each failure

IB1c. Extent of degradation due to each failure (e.g. graceful degradation, reconfiguration, and self-repair)

IB1d. Duration of each degradation

IB2. Integrity

Failures that affect the integrity of the system are those in which the computer is prevented from proceeding because loss of information precludes the computation of a valid result. When a failure does occur, the nature of the mission dictates the necessity of recovering outstanding requests or computations in progress. A requirement may specify the degree to which efforts should be taken to assure data integrity. In AFWET, trajectory and lethality computations should be completed without loss of information. On the other hand, an element that is not engaged in combat with another element might sustain a temporary loss of position without affecting the mission. The cost of recovering from every possible loss of state can be enormous with frequent rollback points causing a significant degradation in performance. The requirements might wisely place a limit on the cost of recovery by identifying states that are not critical enough to warrant a full rollback and recovery exercise.

IB2a. Definition and classification of failures that cause loss of state

IB2b. Probability of each failure

IB2c. Cost of recovery of state

IC. Security

Most of security is arguably in the realm of design, since it pertains to specific means by which the reliability of the system may be enhanced. However, there appear to be two areas in which security may be an appropriate requirement. The first is physical security, which may include, for example, all military standards for pressurized cable, disconnectable terminals, safes for storing classified tapes and disks, and even the criteria for destroying or reusing such storage media. The operational category includes any method that must be used to cipher, modularize, limit transmission or otherwise effect how or where sensitive information will be available. Note that the above physical considerations for reuse of tapes are in some ways part of the operational category, since it is well known that disks, for example, even when erased or overwritten a dozen times, can be made to reveal their original (e.g. classified) information with specialized signal differentiating equipment.

IC1. Physical (e.g. gates, locks, safes, etc.)

IC2. Operational--protection of integrity of information

ID. Operating constraints

ID1. Frequency and duration of use

Both the frequency and duration of use are not only important to know from a staffing and maintenance point of view, but also from the standpoint of available resources. If the computer equipment is, for example, only used as part of the target system for a limited time period, it might be provided as a general facility at other times. Conversely, an already-existing facility may be adequate for supporting the needs of the target system. In a satellite probe, it might be very important to know that a computer module could be connected to some network and used for computational assistance when not operating in its primary capacity.

ID2. Control (e.g. remote, local, or not at all)

Control is another important operating constraint in many systems. An unmanned remote facility cannot be restarted by personnel if a failure occurs. Depending on the ability for personnel to reach the remote site, the equipment may need sophisticated automatic restart and even reconfiguration capabilities. Preventive maintenance may also not be possible in inaccessible locations such as in satellite or deep sea probes. In addition, the proximity of the remote

facility may affect the nature of interaction required, since distant space probes experience significant transmission delays due to speed-of-light limitations.

ID3. Staffing requirements

IE. Physical constraints (e.g. size, weight, power requirements, temperature, humidity, portability, ruggedness)

The physical constraints requirements are intended to include all factors relating to the physical placement of the equipment in the field. In the AFWET system, the nature of the pods connected to the wings of the aircraft placed limits on the size, weight, power requirements, and ruggedness of the equipment that could be placed onto the pods. In some applications, even the camouflaging of the cabinets might be an important physical constraint.

II. System development, evolution, and maintenance

In many organizations, the plan for the development, evolution, and maintenance of the target is a separate document from the requirements, since a development plan is considered to be a statement of how the requirements will be carried out. On the other hand, in many instances, large computer systems are requested and paid for by one group and developed and delivered by another. Constraints on the magnitude and cost of the development effort may very well be considered requirements to the group paying the bill. In this section, we discuss various categories relating to the life-cycle plan for the target system.

IIA. Kind of development

The development of target systems can be divided into two gross categories: those efforts that are directed toward a single delivery date at which time the completed operational system will be furnished, and those efforts which plan to deliver working subsets of the requirements for evaluation in the field before embarking on a more complete version. The single full-scale effort is often itself iterative ([Conn 80]), but the early versions of such a system are not intended for use by the customer. Prototyping may be required in time-critical situations where delivering any kind of working system will fulfill an immediate need. Similarly, in state-of-the-art projects, careful analysis of a system shell may be needed to evaluate human factors and to clarify the requirements. Many software systems are almost always iterative. For example, operating systems are usually updated on a regular basis throughout their useful lives.

IIA1. Single full-scale development

IIA2. Iterative with prototyping

IIB. Scale of effort

The scale of effort is an essential factor in establishing requirements for the development of a target system. In iterative efforts, in which many prototypes or versions are envisioned, resources should be allocated for personnel, equipment, and overhead associated with the development of each version. When development time is included as a requirement, then an evaluation can be made assessing the feasibility of completing the stated goals within the proposed time frame. If an extended or advanced development is foreseen for one or more iterations, this information should also be incorporated into the requirements. Finally, each version should have a plan for delivery and installation. When equipment is to be installed in aircraft or naval vessels, delivery and installation may be a complex technical endeavor. And when numerous installations are already in the field, the requirements may need to call for special procedures for handling the complicated logistics of updating each installation.

For each iteration:

IIB1. Development time

IIB2. Development resources

IIB2a. Personnel

IIB2b. Equipment

IIB2c. Cost

IIB3. Delivery and installation (e.g. packaging, shipment, assembly and test equipment)

IIC. Methodology

IIC1. Quality control standards

"Methodology" includes all the management techniques and procedures that assure the success of the project. Quality control is meant to address software as well as hardware, including the current ideas on top-down, structured, provable, modularized, etc. software. Many organizations now include standardization enforcement software in their compilers and assemblers.

IIC2. Milestones and review procedures (including feasibility studies)

Milestones and review procedures track and evaluate partially completed systems. The milestone is an identifiable stage of completion, which can be used to determine whether parallel efforts are progressing on time with respect to one another. Review procedures are used by the developers themselves to assess the current progress of an effort. The

contracting agency may request a feasibility study for implementing a portion of or the entire system. In this case, a milestone might reflect the point at which the feasibility has been proven, thus enabling the initiation of serious subsystem design efforts.

IIC3. Acceptance criteria (e.g. benchmarks)

While a completed system is supposed to in all ways fulfill the requirements, the acceptance criteria identify specific tests and evaluation factors by which the developed system can be judged. Traditionally, the acceptance criteria are the "teeth" in the contract, against which disputes are settled. Since a target system can almost never be exhaustively tested, it is critical that the acceptance tests cover every significant combination of functions or activities which the system is supposed to carry out successfully.

IID. Priority and changeability

The priority and changeability category recognizes that requirements writers may need some way to incorporate flexibility into the requirements. It may be very important that some requirements be carried out, while others may represent "gold plating." When other constraints are considered, such as cost, size, training, development time, etc., it may be necessary to drop some of the less critical requirements. A means for ranking requirements or associating some weighting factor to particular facets of a target system would be very useful. This ranking can follow Parnas' modularization based on the likelihood of change ([Parnas72a]). If a designer is to be expected to hide, in some modular organization, system decisions that could easily change, the information about what might be changing must be represented in the requirements. Along the same lines, if a general requirement could be satisfied by more than one entirely different solution, it may be necessary to be able to include detailed requirements for each of the solutions. For example, in AFWET, if the transmission of certain information could be satisfied either by ground cables or by microwave communication, the military specifications for each form of transmission would have to be included in the requirements.

IID1. Establishing relative importance of requirements

IID2. Identifying factors likely to change

IID2a. Ordering by changeability

IID2b. Identification of alternative requirements

IIE. Maintenance

The maintenance category here is specifically meant to exclude the evolutionary software activities that are often classified under "maintenance". The requirements are concerned with the

system's breaking down and having to be brought back to working condition. For the software, the requirements might specify the staffing needed or a contractual agreement for fixing bugs. The document might list the kinds of programs or packages that will be supplied to fix bugs, and, in addition, what kinds of software will be embedded in the system (such as error logging or path counters) to aid in the discovery and tracing of errors. For hardware, it is necessary to know who will carry out both the preventive and repair-oriented maintenance The requirements may need to spell out standards for a minimal set of test points at which the repairing individual may probe and assess the operation of the circuit.

IIE1. Software

IIE1a. Responsibility for fixing bugs

IIE1b. Instrumentation (e.g. check points, audit trails, driver programs, simulators)

IIE2. Hardware

IIE2a. Frequency and duration of preventive maintenance

IIE2b. Responsibility for repair of faults

IIE2c. Test equipment and procedures (e.g. test points)

III. Economic context of system development

Very few projects are undertaken in which cost is no object. Even in extravagant programs, cost tradeoffs are seriously considered. However, satisfactory economic decisions are much more likely to be made if the cost goals and guidelines are spelled out in the requirements document.

IIIA. Cost tradeoffs

Requirements for cost tradeoffs establish guidelines for determining whether existing equipment and software can be satisfactorily incorporated into the target system or whether a new effort is required. Very often this off-the-shelf equipment is not ideal and does not entirely satisfy the requirements in every respect. However, these cost-tradeoff requirements can be used effectively to overrule other requirements if the sacrifice is not too great. Some criteria are needed to indicate just how important the ready-made requirement is and what might be given up to fulfill that requirement.

It is important that the requirements convey the intended philosophy for cost tradeoffs. The military is, in many projects, using an approach which identifies a minimal set of capabilities to which desired features are added until a certain cost level is exceeded. The requirements must be able to convey just which functions are needed at any cost and which are add-ons. Note that some projects

(e.g. computer toys or games) may be almost entirely a design-to-cost consideration. Even the nature of the functions may be relatively unimportant compared to the price at which it can be sold. Most projects fall between the two extremes, and the requirements must be able to indicate where the tradeoffs are to be made.

IIIA1. Utilization of existing technology versus development of new

IIIA1a. Hardware (e.g. CPU's, interfaces, peripheral equipment)

IIIA1b. Software (e.g. operating systems,compilers)

IIIA2. Primary objectives--design-to-cost versus design-to-function

IIIA2a. Established minimal requirements for designated levels of cost

IIIA2b. Relating alternate requirements with costs

IIIB. Cost of iterative system development

Many projects are developed iteratively, whether or not the customer sees the intervening stages or prototypes. And almost all projects have milestones or baselines which indicate the achievement of some level of operation or functionality. Without cost limitations placed on these stages by the requirements, funds could be allocated to project phases in an unbalanced manner, starving, for example, later efforts due to disproportionate expenditures at the beginning of the project. In addition, if prototypes are to be delivered to the customer for interim use or evaluation, these costs should be addressed in the requirements.

IIIB1. Cost of each prototype or milestone

IIIB1a. Development cost

IIIB1b. Cost of delivery of prototype

IIIC. Cost of each instance of target system

The development effort may be directed at producing many similar or identical target systems. Under these circumstances, the development costs will usually be amortized over the entire projected production run. Each instance of the target system in this case will have costs both from materials and from the applicable fraction of the development expenses. Any proposed evolutionary change to the target system after delivery will have to take into account the costs of updating each installation.

6. Human Factors

6.1. Introduction

The psychological factors involved in software engineering are certainly one of the most neglected aspects of the entire discipline. Omission of such considerations from requirements analysis and specification may be a major reason for eventual user dissatisfaction with the delivered product. We conjecture that many of the human factors are of as much importance to the user as the so-called functional requirements. Thus, consideration of the psychological impacts, both of and on the user, in the requirements phase should have a substantial effect in helping to deliver software products which truely meet the needs of the user. Furthermore, an aggressive view of this facet of requirements should have an important impact on the lifetime cost of the entire project.

In the following sections we explore the nature and importance of human factors as related to the requirements of a project. We shall discuss the problems of communication between members of the user organization and software engineers, followed by specific human factors problems dealing with the target system's user interface. We then review some tools from psychology which may be used in attempting to solve some of these problems.

6.2. The Communication Problem

Many of the problems which originate early in requirements analysis can be attributed to lack of communication between the user community and the software engineers. There are several well-known communication problems. Ideas may be expressed in a vague or ambiguous manner, goals may be contradictory or incompletely formulated, and various users may have

differing views of the desired system. Realization that these difficulties exist leads to the conclusion that good requirements analysis must depend upon intensive interaction with the user community at all authority levels, as well as feedback from the software engineers concerning their understanding of the desired system.

There are communication problems which cannot be solved simply by verbal communication, however, regardless of the amount of interaction and feedback involved. In this section we shall discuss the problem of novice versus expert knowledge, and the problem of tacit knowledge. These as-yet-unresearched problems may hold the key to developing systems which are well-engineered for humans.

When considering the members of the user community, we shall continue to use the word "user" in its broadest sense, including all of the members of a user organization who will have any degree of contact with the system at any time during its lifetime. This is certainly a broad class of users, with varying degrees of interest in the project, but we need not distinguish between them at this time.

Kaplan cites a rather disturbing example wherein the designer asks, "What do you want?", to which the user responds, "What have you got?" ([Kaplan 76]). Although this is, or should be, an extreme example of initial interaction, it does point out some of the inherent difficulties in user/analyst dialogues. It also makes a strong case for the need to know the user([Shneiderman 79]). This one principle is of major importance, for when the analyst truely knows the user (actually users at all authority levels), then "What do you want?" will be replaced with a whole series of ideas and questions with which meaningful discussion can begin.

It would be naive to think, however, that the analyst will find it

easy to "get to know" the user. One major problem is that experts "see" quite differently from novices ([Kaplan 76]), as has been demonstrated in studies such as the famous one involving chess masters and novices ([Simon 70]). In requirements analysis, the user is an expert in the application domain, the analyst is an expert in software engineering, and each is a novice in the domain of the other!

This phenomena of seeing differently is partially explained by the compact and complex structure of the experts' knowledge. Experts have more points of entry into their semantic structures, and they form abstractions at a higher level than novices. This means that they have the ability to make use of a number of different representation schemes for mentally working with the same information. Thus, an expert might be able to make use of a picture, a sketch, a map, or even a cardboard model, whereas a novice might find the picture to be the only meaningful representation ([Weiser 79]). So we must conclude that the choice of model presentation is a critical factor to be considered when dealing with novice users. In general, the process of establishing a common domain of discourse should be the first matter of attention in a user/analyst dialogue. Also, this has to take place at all levels within the user organization, because of the different views of the users.

We know that people, whatever their organizational status, have detailed and highly developed internal models of their working environments ([Kaplan 76]). They understand what they do, how they accomplish their duties, and with whom or what they interact in their performance of daily tasks (we speak, of course, of people with a general level of work-related competence.) Regardless of their level of expertise, however, people know more than they can ever tell ([Polanyi 67]). When this "tacit

knowledge" concerns a desired software product, we are often at a loss as to how to bring forth this information. The problem is more than just a vagueness on the part of the user concerning desired functions for a proposed system, because tacit knowledge is not describable by the user. Some tacit knowledge will always exist, but much can be brought out and made explicit.

Polanyi presents this example to illustrate the existence of tacit knowledge: We are all experts at recognition of familiar faces, yet how many features of a familiar face can you give specific details about? This is very difficult, even for faces with which you may be intimately familiar. However, police artists have developed methods which allow them to produce composite sketches of remarkable quality.

The idea of tacit knowledge is certainly not new. William James, in discussing what he called the "fringe" ([James 92]), said that

> "Every definite image in the mind is steeped and dyed in the free water that flows round it. With it goes the sense of its relations, near and remote, the dying echo of whence it came to us, the dawning sense of whither it is to lead. The significance, the value, of the image is all in the halo or the penumbra that surrounds and escorts it..."

Thus, we must probe, define and refine this fringe in order to discover some of the tacit knowledge contained therein.

Since the users are incapable of expressing their tacit knowledge, we should consider experimental techniques for revealing some of it. For now we simply note that people have an innate ability for nonverbal communication, primarily with themselves. People have the capacity to assume an "as if" stance ([Kaplan 76])--they have the ability to assume roles and pretend. This means that one should expect to be able to produce worthwhile results from studies and experiments concerning human factors engineering. This has certainly proved to be the case in

scientific investigations of programming languages ([Basili and Reiter 79], [Dunsmore and Gannon 79], [Shneiderman 79], [Shneiderman 80]). We will have more to say on the subject of experimentation in the section on psychological tools.

In summary then, the apparent requirements will vary depending upon the view of the user. Regardless of his status within an organization, each user will have a well-developed internal model of the environment and his functions within it. Some of this information is readily available in the form of immediate needs (however vaguely they may be expressed), but some is tacit knowledge and may have to be determined experimentally.

6.3. The Target System's User Interface

In discussing the user interface requirements for a system, it is essential to differentiate between upper-echelon management and the end-user ([Mittermeir 79], [Mittermeir 80]). Their views of the system at this level are sure to be different. Management will be primarily concerned with the system's functional requirements, constraints, development schedule, and cost. The end-user, however, may take such factors as "correctness" for granted; he or she must work with the final product, and wants a system which provides a comfortable environment, not a hostile one.

The AFWET system requires displays, and so we will concentrate on displays as a good example of various user-interface issues. Luxenberg and Kuehn ([Luxenberg and Kuehn 69]) note that

> "It is essential to display design that standard human factors requirements be satisfied. This covers a broad range of topics such as perception, comprehension, viewing environment, psychological factors, and operator comfort."

These issues are of great importance if the system is to be acceptable to the user community. Thus, in a requirements document it is not enough

to simply specify that the display equipment will consist of certain kinds of devices, which is all that was done in the AFWET document. The details which seem to be most needed are given by Luxenberg and Kuehn and are shown in Table 2.

The information required in each of these categories and subcategories, however, is much more than just specifications in terms of some absolutes or generalities, e.g., response time must be 3 seconds, or response must be real time. The requirements document should have proposals for testing the acceptability of the factors with the end-users. Thus, we are immediately lead into the general area of testability and the questions which naturally arise concerning what is or is not a testable requirement. Nevertheless, if there is to be a display which queries a database, for example, then the requirements for response time should also include either solid reasons for specifying a certain value, or proposed experiments to determine what the needs of the end-user really are and how this will be validated, both from the standpoint of verifying what the need is, as well as checking that the final product meets the need. This practice should be applied liberally across any parts of the specification which deal with the broad area of human factors.

Thus, requirements for AFWET should contain proposals for pilot studies of: (a) the best symbology to use for the displays; (b) the qualities desired in such device specifics as contrast, resolution, and flicker; (c) the best means to display a "kill". There should also be proposals to study whether or not the display needs to run "in real time", as stated in the AFWET document. Even a small difference in the amount of acceptable delay can make a tremendous difference to the system's designers, and it is all too easy to accept serious constraints on a user's

word, without questioning the real need for them. It certainly seems plausible that observers could get as much information from delayed or replayed tests as from those seen in real time. In this case, probably the crucial factor is the extent to which the observers of the displays participate in the test as commanders, but this is nowhere mentioned in the requirements document.

6.4. Psychological Tools

Some of the problems of user/analyst interaction have been highlighted in the previous sections. We believe that aggressive work on the human factors of a project will not only help alleviate some of the inherent communication problems, but also provide a sound basis for a project which is manageable in terms of schedule, cost functionality, and human acceptability.

Besides the obvious need for much interaction and discussion in order to speak on common terms, what else can be done to help alleviate this communication problem? Winograd discusses three different domains of discourse and suggests that the terms used in the subject domain be those familiar to the user ([Winograd 79]). Other ideas presented by Kaplan ([Kaplan 76]) suggest that we can make use of the users' abilities to assume roles, to mentally validate/reject "what if" conditions, and to become involved in the entire process of design. He suggests that the use of simple models works better from the viewpoint of the user than do complicated or elaborate models. This confirms what we already know about expert versus novice knowledge structures. So, once a dialect has been established, the software engineer may begin planning experiments, pilot tests, and other interactions with actual end-users in an attempt to bring forth tacit knowledge which may play an integral role in the

functioning of the desired product.

The early phases of requirements anlysis should concentrate upon dissolving the differences discussed earlier in the ways in which experts and novices see. Definitions (based upon the users' perspective), intensive discussions, and notations should help make problem areas explicit, remove bias, and thus add to the fringes of both user and analyst. Then the use of models, quick prototypes, graphic aids, and other forms of nonverbal communication should be encouraged, as these give the user something which can be "indwelled" ([Polanyi 67]), i.e., internalized. This process of indwelling is most important, for it is the only way to really know something. The "one picture is worth ten thousand words" idea may sound too simple, but the replacement of words by actions can give the user a better understanding of what is being developed. Thus, by having something to indwell, the user will have the ability to make a comparison with his internal model, which already exists. This could not be done, and certainly is not done, with any of the static, formal notations currently in use for the specification of requirements.

The use of models, and their associated tests with actual end-users, should become a part of the planned system development from the very beginning. These pilot tests must be used for a sufficient period of time to allow the user to get beyond any difficulties of novelty. They must also be repeated many times after there are no more learning problems, because the nature of "participant behavior" is characterized by the fact that the participant considers a number of hypotheses on each trial and can only reject some, but not all, of those which are not consistent with his internal models ([Posner 73]). Furthermore, users can more readily reject undesirable qualities than affirm desirable ones,

probably because many of the desirable properties are part of their tacit knowledge.

Finally, studies should be planned to further define/refine human needs. Brooks ([Brooks 80]) states advantages of behavioral/psychological studies which are important here for two reasons. First, we can affirm/refute any behavioral assumptions which have been made. Second, such studies give quantitative information on the relative effectiveness of various techniques, thus giving us a solid basis for the selection of new tools, new features, and new areas of concern. Careful selection of the studies to be made can help reduce costs for the entire project by confirming at an early stage that any of a number of quality-control attempts are, or are not, successful. We should be most concerned with the ideas of simplicity, psychological acceptability, the "engineering out" of errors, and the bounds on human performance ([Shneiderman and Mayer 79], [Shneiderman 79], [Shneiderman 80]). Scientific experimentation during the requirements phase can help assure that the developing product will meet any of a number of such goals, and at a lesser cost than if these are ignored until later in the project lifetime.

There may certainly be economic considerations and developmental time limitations to restrict the amount of experimentation involved in a particular project. Time constraints on the end-users may also be a factor. However, the nature of human factors is such that they are very amenable to a type of requirement which specifies that a study or experiment be used to further define some quality of the end-users' environment. The life-cycle of a project may easily be long enough so that, with good modularization of requirements, such studies can proceed in parallel with some of the other work on the project.

Our section on human factors is intended, like the rest of this document, to serve as a checklist for topic inclusion. Many of the suggested areas should be cross-referenced with proposals, milestones, and experiments as in the section on system development. Note that the issues raised in this section should follow from the broadest possible interpretation of human factors--we intend for the requirements to consider a wide range of psychological factors, e.g., user acceptability, motivation, and the work environment.

We have been deliberately vague about the types of pilot tests, models, and other nonverbal communication tools the engineer may find of value. This is an area where much research needs to be done to discover what types of prototypes work best for the desired interaction between user and designer. Kaplan has performed some studies with architects and users, and his results indicate that simplicity helped avoid much confusion on the part of the user ([Kaplan 76]).

Another example concerns the use of the "operational" requirements specifications mentioned in Section 4. The requirements consist of an executable model of the proposed system interacting with its environment, and this model could be exercised interactively to provide demonstrations to users. This is a promising direction--since functional and performance requirements at all levels can be incorporated into the formal specification ([Zave 79a], [Zave 79b]), the model could provide a basis for conducting many of the experiments proposed in the requirements document. The next question, however, is how can system behaviors be communicated to the user? What tests of the system should be performed? How can the information be made suitable for indwelling by the user? These are issues relating to nonverbal communication which deserve immediate attention.

7. Conclusion

This report is by no means a complete survey of current knowledge on requirements. Some of the best-known approaches have been given short shrift (although they have already been widely reviewed). We have made no attempt to survey tools, even though it is apparent that automated database facilities for requirements information, however primitive, may be tremendously helpful.

The significance of this article, in our view, is that we have included the "forgotten" areas of requirements: process-oriented as well as data-oriented requirements, nonfunctional as well as functional requirements, and human factors. We have stressed the newest and (to us) most promising approaches, over the familiar and (to us) inadequate ones.

We believe that the problem of deriving good requirements can be solved in three stages. The first concerns discovering, understanding, and describing informally the users' requirements (interpreting "user" in its broadest sense). The second involves constructing a conceptual model which integrates and consolidates different user views. The third consists of specifying a system to meet the requirements in an executable language, and validating that specification.

Although we have categorized specification languages on the basis of system types, they also fit some of these stages better than others. PAISLey is a good candidate for an executable specification language, but a structured data model expressed using semantic nets many make the best all-around conceptual model. And for first-stage description of users' needs, application-specific, user-oriented languages are clearly called for. It is our belief that a general

framework for such languages can be developed on the basis of a case-structured syntax. These are the directions which researchers will be pursuing in the near future.

Acknowledgement

The research reported on here is partially supported by the U.S. Air Force under Contract AFOSR-77-3181B, and by the U.S. Army under Contract DASG60-80-C-0024.

References

[Air Force 74]
Air Force, U.S., "Air Force Weapons Effectiveness Testing (AFWET) Instrumentation System," RDD Exhibit No. PSVE 74-3, Air Proving Ground Center, Eglin Air Force Base, Florica, 1974.

[Alford 77]
Alford, Mack, "A Requirements Engineering Methodology for Real-Time Processing Requirements," IEEE Transactions on Software Engineering 3, January 1977, pp. 60-69.

[Astrahan et. al. 76]
Astrahan, M. M., et. al., "System R: Relational Approach to Database Management," ACM Transactions on Database Systems 1, June 1976, pp. 97-137.

[Backus 78]
Backus, John, "Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs," Communications of the ACM 21, August 1978, pp. 613-641.

[Basili & Reiter 79]
Basili, Victor R., and Reiter, Robert W., Jr., "An Investigation of Human Factors in Software Engineering," Computer 12, December 1979, pp. 21-38.

[Belady & Lehman 79]
Belady, L. A., and Lehman, M. M., "The Characteristics of Large Systems," Research Directions in Software Technology, Peter Wegner (ed.), M.I.T. Press, 1979, pp. 106-138.

[Bell et. al. 77]
Bell, Thomas, Bixler, David, and Dyer, Margaret, "An Extendable Approach to Computer-Aided Software Requirements Engineering," IEEE Transactions on Software Engineering 3, January 1977, pp. 49-60.

[Boehm 73]
Boehm, Barry W., "Software and Its Impact: A Quantative Assessment," Datamation 19, May 1973, pp. 48-59.

[Boehm 79]
Boehm, Barry, Software Engineering Economics, Prentice Hall, to appear.

[Brinch Hansen 78]
Brinch Hansen, Per, "Distributed Processes: A Concurrent Programming Concept," Communications of the ACM 21, November 1978, pp. 934-941.

[Brooks 80]
Brooks, Ruven E., "Studying Programmer Behavior Experimentally: The Problem of Proper Methodology," Communications of the ACM 23, April 1980, pp. 207-213.

[Conn 80]
Conn, Alex Paul, "Maintainance: A Key Element in Computer Requirements Definition," Proceedings COMPAC, 1980, to appear.

[Conner 79]
Conner, M., Process Synchronization by Behavior Controllers, M.S. Thesis, C. S. Department, University of Texas at Austin, 1979.

[Davis & Vick 77]
Davis, Earl W., and Vick, Charles R., "The Software Development System," IEEE Transactions on Software Engineering 3, January 1977, pp. 69-84.

[Gunsmore & Gannon 77]
Gunsmore, H. E., and Gannon, J. D., "Data Referencing: An Empirical Investigation," Computer 12, December 1977, pp. 50-59.

[Hoare 78]
Hoare, C. A. R., "Communicating Sequential Processes," Communications of the ACM 2, August 1978, pp. 666-677.

[Horning & Randell 73]
Horning, J. J., and Randell, B., "Process Structuring," Computing Surveys 5, January 1973, pp. 5-29.

[James 92]
James, William, Psychology: The Briefer Course, Harper, 1892, pp. 30-37 and 108-110.

[Jones & Schwarz 80]
Jones, A. K., and Schwarz, P., "Experience Using Multiprocessor Systems -- A Status Report," Computing Surveys 12, June 1980, pp. 121-165.

[Kaplan 76]
Kaplan, Stephen, "Participation in the Design Process," Psychological Perspectives on Environment and Behavior: Conceptual and Empirical Trends, S. Stokes (ed.), Plenum, 1976.

[Lehman 80]
Lehman, M., "Programs, Programming, and the Software Life Cycle," Proceedings of the IEEE 68, September 1980, to appear.

[Luxenburg & Kuehn 68]
Luxenburg, H. R., and Kuehn, R. L., Display Systems Engineering, McGraw Hill, 1968.

[Mittermeir 79]
Mittermeir, Roland, "Application of Database Design Concepts to Software Requirements Analysis," TR DA 79/11/01, Institut fuer Digitale Anlagen, Technische Universitaet Wien, Wien 1979.

[Mittermeir 80]
Mittermeir, Roland, "Requirements Analysis: Top Down or Bottom Up," TR DA 80/02/02, Institut fuer Digitale Anlagen, Technische Universitaet Wien, Wien 1980.

[Parnas 72a]
Parnas, D. L., "On the Criteria to be Used in Decomposing Systems into Modules," Communications of the ACM 15, December 1972, pp. 1053-1058.

[Parnas 72b]
Parnas, D. L., "A Technique for Software Module Specification with Examples," Communications of the ACM 15, May 1972, pp. 330-336.

[Parnas 77]
Parnas, D. L., "Designing Software for Ease of Extension and Contraction," IEEE Transactions on Software Engineering 5, March 1979, pp. 128-137.

[Peterson 77]
Peterson, James L., "Petri Nets," Computing Surveys 9, September 1977, pp. 223-252.

[Polanyi 67]
Polanyi, Michael, The Tacit Dimension, Anchor Books (Doubleday), 1967, pp. 3-25.

[Posner 77]
Posner, Michael I., Cognition: An Introduction, Scott, Foresman and Company, 1973, pp. 61-62.

[Ross 77]
Ross, Douglas, "Structured Analysis (SA): A Language for Communicating Ideas," IEEE Transactions on Software Engineering 3, January 1977, pp. 15-34.

[Ross & Schoman 77]
Ross, Douglas, and Schoman, Kenneth E. Jr., "Structured Analysis for Requirements Definition," IEEE Transactions on Software Engineering 3, January 1977, pp. 6-15.

[Roussopoulous 79]
Roussopoulous, Nicholas, "CSDL: A Conceptual Schema Definition Language for the Design of Data Base Applications," IEEE Transactions on Software Engineering 5, September 1979, pp. 477-487.

[Shneiderman 79]
Shneiderman, Ben, "Human Factors Experiments in Designing Interactive Systems," Computer 12, December 1979, pp. 9-20.

[Shneiderman 80]
Shneiderman, Ben, Software Psychology: Human Factors in Computer and Information Systems, Winthrop Publishers, Inc. 1980.

[Shneiderman & Mayer 79]
Shneiderman, Ben and Mayer, Richard, "Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results," Int. Journal of Computer and Information Sciences 8, March 1979, pp. 219-238.

[Simon 69]
Simon, Herbert, The Sciences of the Artificial, M.I.T. Press, 1969.

[Smith & Smith 79]
Smith, John Miles, and Smith, Diane C. P., "A Database Approach to Software Specification," Proceedings Software Development Tools Workshop, Pingree Park, Colorado, May, 1979, to appear.

[Smolar 79]
Smolar, Stephen W., "Using Applicative Techniques to Design Distributed Systems," Proceedings Specifications of Reliable Software Conference, Cambridge Mass., April 1979, pp. 159-161.

[Smolar]
Smolar, Stephen W., "Applicative and Functional Programming," Software Engineering Handbook, C. V. Ramamoorthy and Charles Dick (eds.), Prentice-Hall, To appear.

[Thayer 77]
Thayer, T. A., "Understanding Software Through Analysis of Empirical Data," TRW Software Series, TRW-SS-77-04, Nov 1977.

[Teichroew , Hershey 77]
Teichroew, D., and Hershey, E. A. III, "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Systems," IEEE Transactions on Software Engineering 3, January 1977, pp.41-48.

[Weiser 79]
Weiser, Mark David, Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Model, Ph.D. Thesis, University of Michigan, 1979.

[Wingard 79]
Wingard, Terry, "Beyond Programming Languages," Communications of the ACM 22, July 1979, pp. 391-401.

[Yeh et. al. 79]
Yeh, Raymond T., "Systematic Derivation of Software Requirements Through Structured Analysis," Computer Science SOFTEC-79, University of Texas at Austin, September 1979.

[Yeh & Zave 79]
Yeh, Raymond T., and Zave, Pamela, "Specifying Software Requirements," Proceedings of the IEEE 68, September 1980, to appear.

[Zave 79b]
Zave, Pamela, "A Comprehensive Approach to Requirements Problems," Proceedings COMPSAC, Chicago Ill., November 1979, pp. 117-122.

[Zave 79c]
Zave, Pamela, "Formal Specification of Complete and Consistent Performance Requirements," Proceedings Texas Conference on Computing Systems, Dallas Tx., November 1979, pp. [illegible] - 4:-2[illegible].

[Zave ??]
Zave, Pamela, "The Operational Approach to Requirements Specification for Embedded Systems," University of Maryland Computer Science Technical Report, in preparation, 1980.

[Zave & Yeh ??]
Zave, Pamela, and Yeh, Raymond T., "Executable Requirements for Embedded Systems," submitted for publication.

[Zelkowitz et. al. 79]
Zelkowitz, M. V., Shaw, A. C., and Gannon, J. D., Principles of Software Engineering and Design, Prentice Hall, 1979, pp. [illegible].

```
def  TEST-ELEMENT:
     sub  PLANE, SHIP, GROUND-DEFENSE-POSITION, TARGET
     com  ROLE
     end
def  PLANE:
     sub  B-52G, F-4C
     com  AIR-POSITION, WEAPONS
     end
def  SHIP:
     com  SURFACE-POSITION, WEAPONS
     end
def  TANK:
     com  SURFACE-POSITION, WEAPONS
     end
def  GROUND-DEFENSE-POSITION:
     com  SURFACE-POSITION, WEAPONS
     end
def  TARGET:
     sub  BRIDGE, DEPOT
     com  SURFACE-POSITION
     end
def  AIR-POSITION:
     com  SURFACE-POSITION, ALTITUDE
     end
def  BATTLEFIELD:
     com  TEST-ELEMENT*
     end
def  TEST:
     com  FRAME*
     end
def  FRAME:
     com  TIME, TEST-ELEMENT*
     end
```

Table 1. AFWET type definition.

Table 2. A major step in display design is determination of specifications for the following:

A. Data Rates and Response Times
   1. updating response time
   2. rates of change of display data
   3. display access time
   4. display request rates

B. Amount of Data
   1. amount of display information
   2. number of display units
   3. display sizes
   4. audience size

C. Types of Display
   1. coding
   2. symbology
   3. display formats

D. Visibility
   1. luminance
   2. ambient lighting
   3. contrast
   4. resolution

E. Quality
   1. accuracy
   2. distortion
   3. flicker

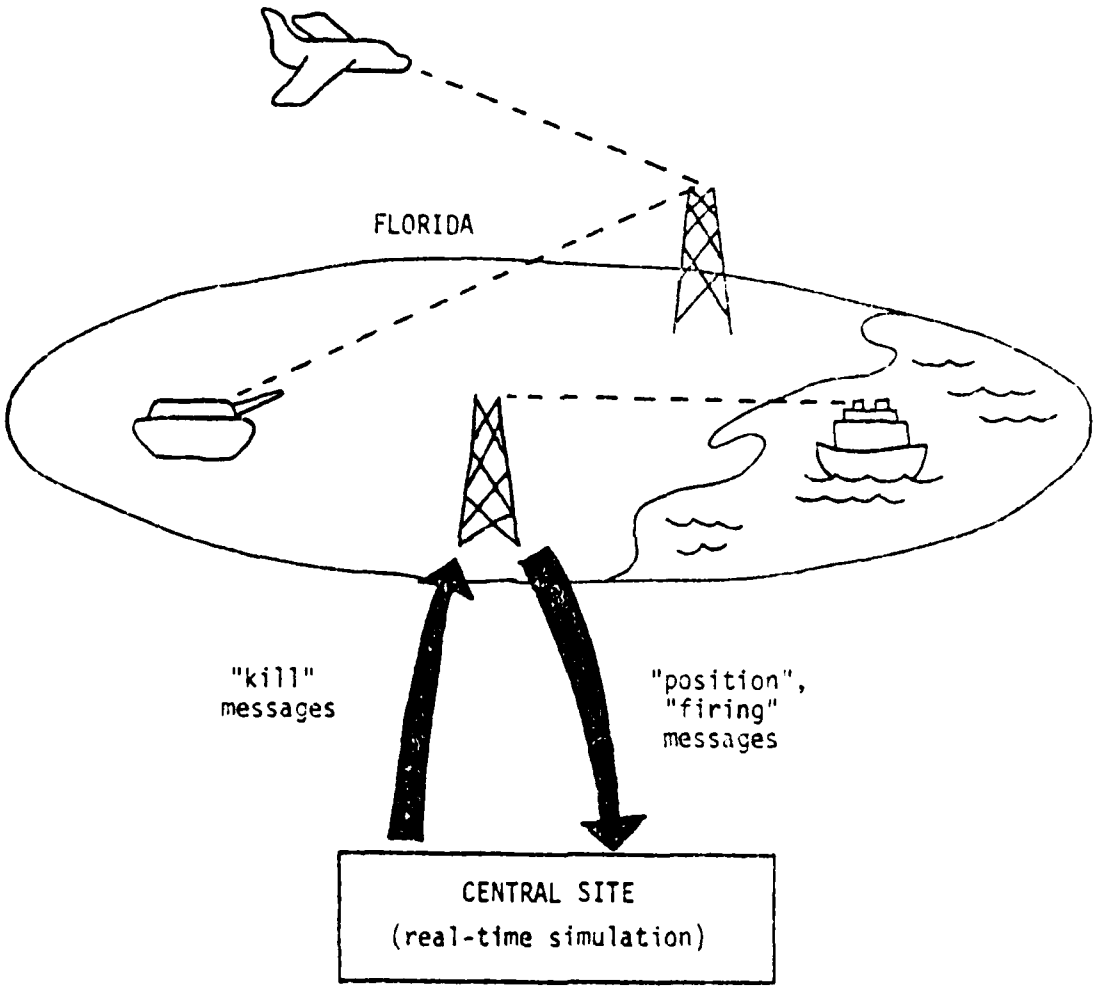


Figure 1 The AFWET system.

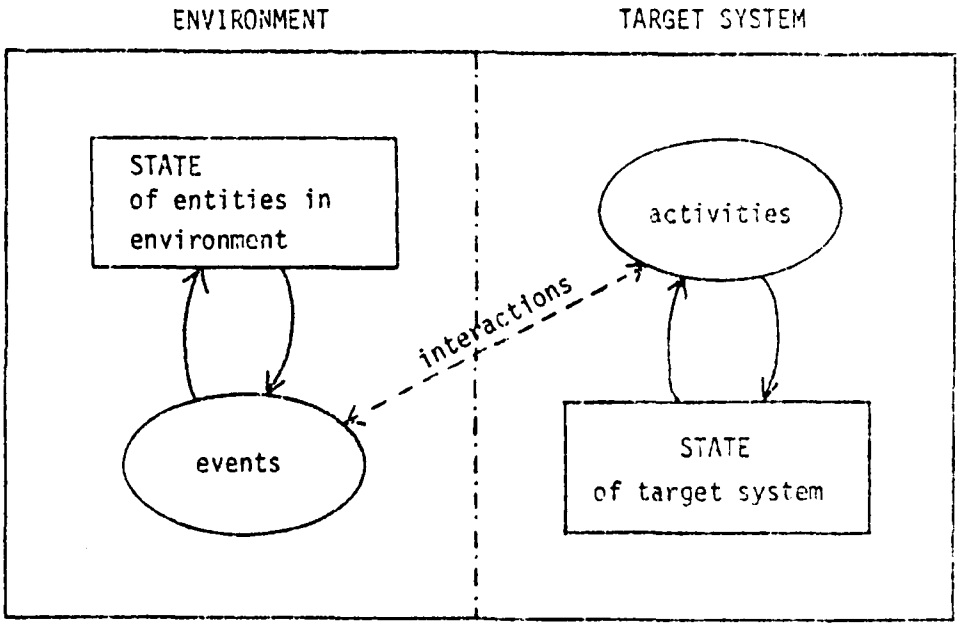


Figure 2. A conceptual model.

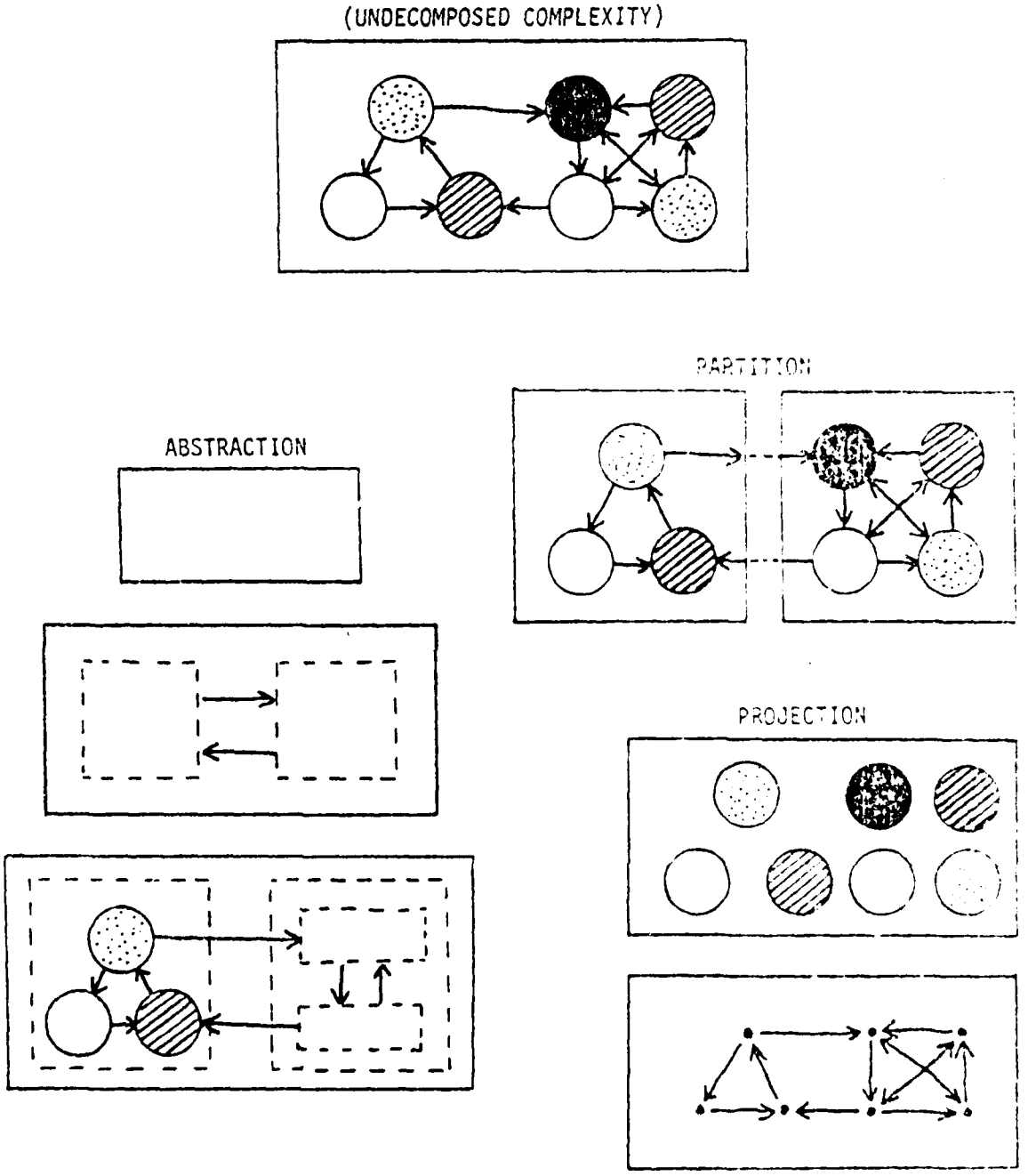


Figure 3. Three ways to decompose complexity.

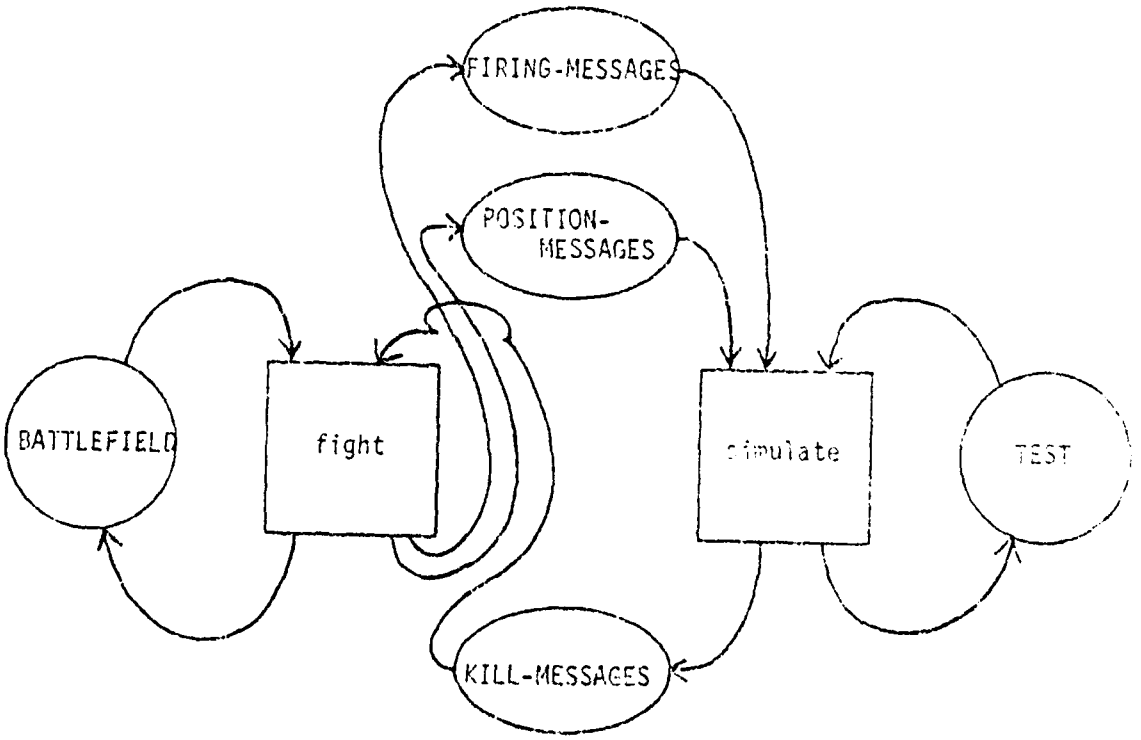


Figure 4. AFWET dataflow.

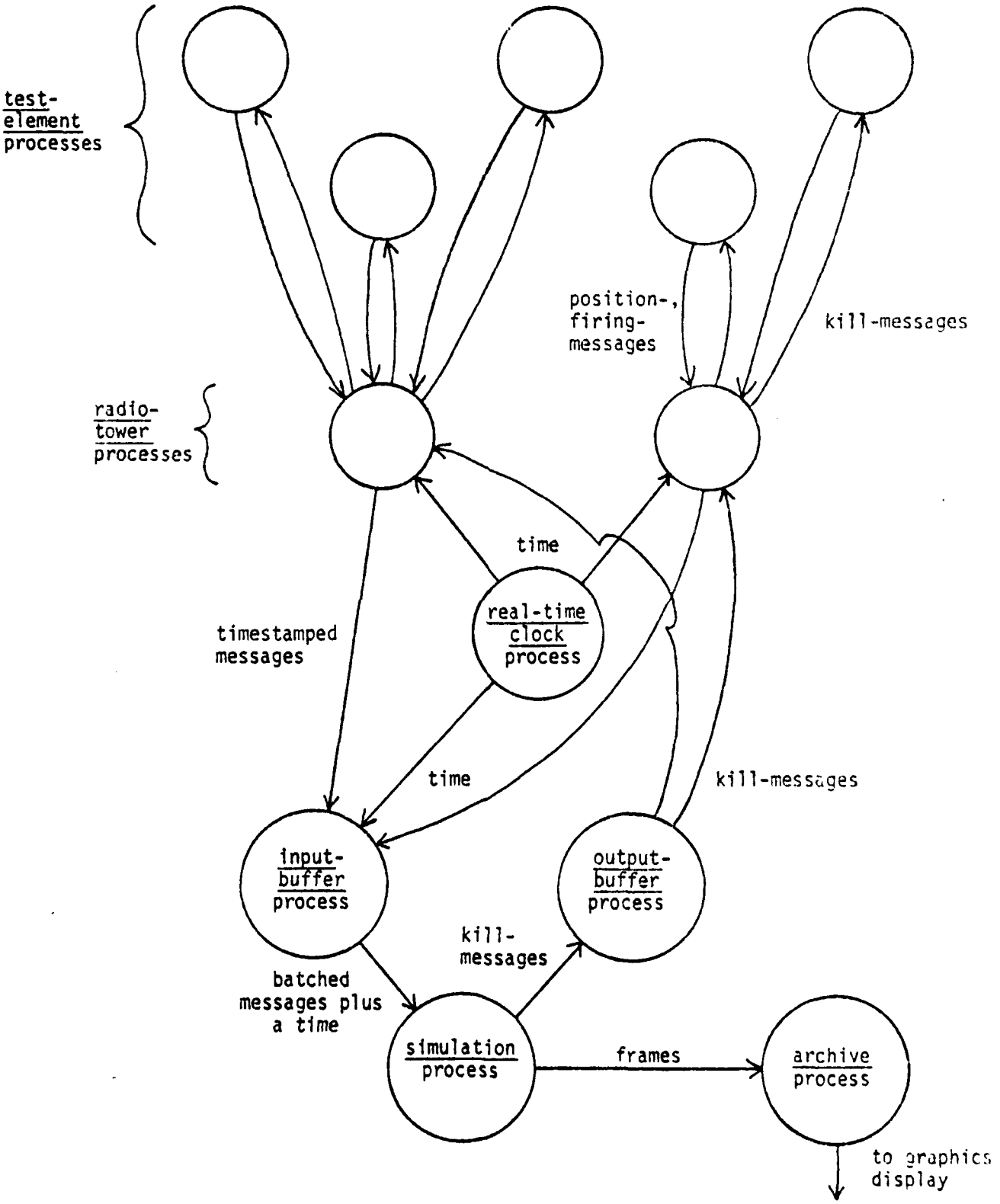


Figure 5. Processes of the AFWET system.

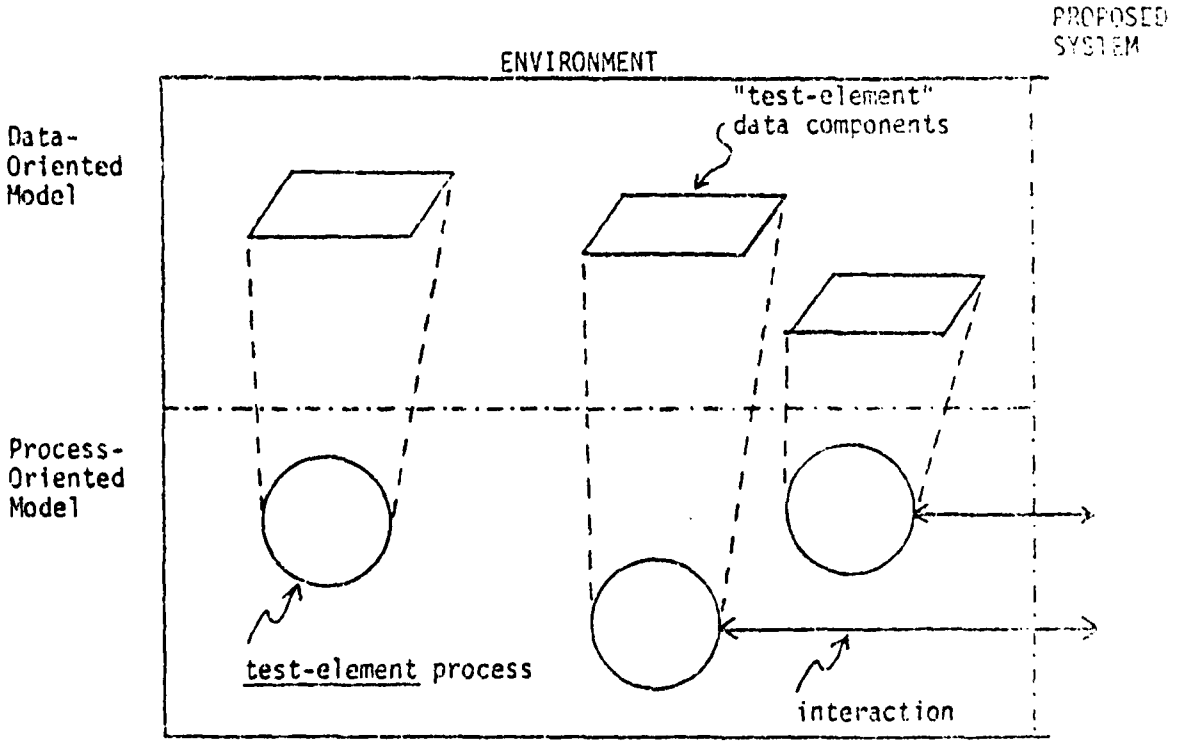


Figure 6. The AFWET environment model, projected onto process-oriented and data-oriented views.


DATE
ILME